ABSTRACT

In unit testing, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are memory graphs. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining symbolic and concrete execution, and more specifically, using such a combination to generate test inputs to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of such test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Verification

Keywords: concolic testing, random testing, explicit path model-checking, data structure testing, unit testing, testing C programs.

1. INTRODUCTION

Unit testing is a method for modular testing of a program's functional behavior. A program is decomposed into units, where each unit is a collection of functions, and the units are independently tested. Such testing requires specification of values for the inputs (or test inputs) to the unit. Manual specification of such values is labor intensive and cannot guarantee that all possible behaviors of the unit will be observed during the testing.

In order to improve the range of behaviors observed (or test coverage), several techniques have been proposed to automatically generate values for the inputs. One such technique is to randomly choose the values over the domain of potential inputs [4,8,10,22]. The problem with such random testing is two fold: first, many sets of values may lead to the same observable behavior and are thus redundant, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [21].

One approach which addresses the problem of redundant executions and increases test coverage is symbolic execution [1,3,9,23,24,27,28,30]. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an execution path. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking [15]. Unfortunately, for large or complex units, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

To the best of our knowledge, Larson and Austin were the first to propose combining concrete and symbolic execution [17]. In their approach, the program is executed on some user-provided concrete input values. Symbolic path constraints are generated for the specific execution. These constraints are solved, if feasible, to see whether there are potential input values that would have led to a violation along the same execution path. This improves coverage while avoiding the computational cost associated with full-blown symbolic execution which exercises all possible execution paths.

Godefroid et al. proposed incrementally generating test inputs by combining concrete and symbolic execution [11]. In Godefroid et al.’s approach, during a concrete execution, a conjunction of symbolic constraints along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate further test inputs which would direct the program along alternative paths. Specifically, they systematically negate the conjuncts in the path constraint to provide a depth first exploration of all paths in the computation tree. If it is not feasible to solve the modified constraints, Godefroid et al. propose simply substituting random concrete values.

A challenge in applying Godefroid et al.’s approach is to provide methods which extract and solve the constraints generated by a program. This problem is particularly complex for programs which have dynamic data structures using
### Abstract

In unit testing, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are memory graphs. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining symbolic and concrete execution, and more specifically, using such a combination to generate test inputs to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of such test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.
pointer operations. For example, pointers may have aliases. Because alias analysis may only be approximate in the presence of pointer arithmetic, using symbolic values to precisely track such pointers may result in constraints whose satisfiability is undecidable. This makes the generation of test inputs by solving such constraints infeasible. In this paper, we provide a method for representing and solving approximate pointer constraints to generate test inputs. Our method is thus applicable to a broad class of sequential programs.

The key idea of our method is to represent inputs for the unit under test using a logical input map that represents all inputs, including (finite) memory graphs, as a collection of scalar symbolic variables and then to build constraints on these inputs by symbolically executing the code under test.

We first instrument the code being tested by inserting function calls which perform symbolic execution. We then repeatedly run the instrumented code as follows. The logical input map \( I \) is used to generate concrete memory input graphs for the program and two symbolic states, one for pointer values and one for primitive values. The code is run concretely on the concrete input graph and symbolically on the symbolic states, collecting constraints (in terms of the symbolic variables in the symbolic state) that characterize the set of inputs that would (likely) take the same execution path as the current execution path. As in [11], one of the collected constraints is negated. The resulting constraint system is solved to obtain a new logical input map \( I' \) that is similar to \( I \) but (likely) leads the execution through a different path. We then set \( I = I' \) and repeat the process. Since the goal of this testing approach is to explore feasible execution paths as much as possible, it can be seen as Explicit Path Model-Checking.

An important contribution of our work is separating pointer constraints from integer constraints and keeping the pointer constraints simple to make our symbolic execution lightweight and our constraint solving procedure not only tractable but also efficient. The pointer constraints are conceptually simplified using the logical input map to replace complex symbolic expressions involving pointers with simple symbolic pointer variables (while maintaining the precise pointer relations in the logical input map). For example, if \( p \) is an input pointer to a struct with a field \( f \), then a constraint on \( p.f \) will be simplified to a constraint on \( f_0 \), where \( f_0 \) is the symbolic variable corresponding to the input value \( p.f \). Although this simplification introduces some approximations that do not precisely capture all executions, it results in simple pointer constraints of the form \( x = y \) or \( x \neq y \), where \( x \) and \( y \) are either symbolic pointer variables or the constant NULL. These constraints can be efficiently solved, and the approximations seem to suffice in practice.

We implemented our method in a tool called CUTE (Concolic Unit Testing Engine, where Concolic stands for cooperative Concrete and symbolic execution). CUTE is available at http://oal.cs.uic.edu/~ksen/cute/. CUTE implements a solver for both arithmetic and pointer constraints to incrementally generate test inputs. The solver exploits the domain of this particular problem to implement three novel optimizations which help to improve the testing time by several orders of magnitude. Our experimental results confirm that CUTE can efficiently explore paths in C code, achieving high branch coverage and detecting bugs. In particular, it exposed software bugs that result in assertion violations, segmentation faults, or infinite loops.

```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
    return 0;
}
```

Figure 1: Example C code and inputs that CUTE generates for testing the function `testme`.

This paper presents two case studies of testing code using CUTE. The first study involves the C code of the CUTE tool itself. The second case study found two previously unknown errors (a segmentation fault and an infinite loop) in SGLIB [25], a popular C data structure library used in a commercial tool. We reported the SGLIB errors to the SGLIB developers who fixed them in the next release.

### 2. Example

We use a simple example to illustrate how CUTE performs testing. Consider the C function `testme` shown in Figure 1. This function has an error that can be reached given some specific values of the input. In a narrow sense, the input to `testme` consists of the values of the arguments \( p \) and \( x \). However, \( p \) is a pointer, and thus the input includes the memory graph reachable from that pointer. In this example, the graph is a list of `cell` allocation units.

For the example function `testme`, CUTE first non-randomly generates `NULL` for \( p \) and randomly generates 236 for \( x \), respectively. Figure 1 shows this input to `testme`. As a result, the first execution of `testme` takes the `then` branch of the first `if` statement and the `else` branch of the second `if`. Let \( p_0 \) and \( x_0 \) be the symbolic variables representing the values of \( p \) and \( x \), respectively, at the beginning of the execution. CUTE collects the constraints from the predicates of the branches executed in this path: \( x_0 > 0 \) (for the `then` branch of the first `if`) and \( p_0 = \text{NULL} \) (for the `else` branch of the second `if`). The predicate sequence \( (x_0 > 0, p_0 = \text{NULL}) \) is called a path constraint.

CUTE next solves the path constraint \((x_0 > 0, p_0 \neq \text{NULL})\), obtained by negating the last predicate, to drive the next execution along an alternative path. The solution that CUTE proposes is \( (p_0 \mapsto \text{non-NULL}, x_0 \mapsto 236) \), which requires that CUTE make \( p \) point to an allocated `cell` that introduces two new components, `p->v` and `p->next`, to the reachable graph. Accordingly, CUTE randomly generates 634 for `p->v` and non-randomly generates `NULL` for `p->next`, respectively, for the next execution. In the second execution, `testme` takes the `then` branch of the first and the second `if` and the `else` branch of the third `if`. For this execution, CUTE generates the path constraint \((x_0 > 0, p_0 \neq \text{NULL}, 2 : x_0 + 1 \neq v_0)\), where \( p_0, v_0, n_0 \), and \( x_0 \) are the symbolic values of \( p \), `p->v`, `p->next`, and \( x \), respectively. Note that CUTE computes the expression
2 \cdot x_0 + 1$ (corresponding to the execution of $f$) through an inter-procedural, dynamic tracing of symbolic expressions.

CUTE next solves the path constraint $(x_0 > 0, p_0 \neq \text{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 = n_0)$, obtained by negating the last predicate and generates Input 3 from Figure 1 for the next execution. Note that the specific value of $x_0$ has changed, but it remains in the same equivalence class with respect to the predicate where it appears, namely $x_0 > 0$. On Input 3, testme takes the then branch of the first three if statements and the else branch of the fourth if. CUTE generates the path constraint $(x_0 > 0, p_0 \neq \text{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 = n_0)$. This path constraint includes dynamically obtained constraints on pointers. CUTE handles constraints on pointers but requires no static alias analysis. To drive the program along an alternative path in the next execution, CUTE solves the constraints $(x_0 > 0, p_0 \neq \text{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 = n_0)$ and generates Input 4 from Figure 1. On this input, the fourth execution of testme reveals the error in the code.

3. CUTE

We first define the input logical input map that CUTE uses to represent inputs. We also introduce program units of a simple C-like language (cf. [20]). We present how CUTE instruments programs and performs concolic execution. We then describe how CUTE solves the constraints after every execution. We next present how CUTE handles complex data structures. We finally discuss the approximations that CUTE uses for pointer constraints.

To explore execution paths, CUTE first instruments the code under test. CUTE then builds a logical input map $I$ for the code under test. Such a logical input map can represent a memory graph in a symbolic way. CUTE then repeatedly runs the instrumented code as follows:

1. It uses the logical input map $I$ to generate a concrete input memory graph for the program and two symbolic states, one for pointer values and another for primitive values.

2. It runs the code on the concrete input graph, collecting constraints (in terms of the symbolic values in the symbolic state) that characterize the set of inputs that would take the same execution path as the current execution path.

3. It negates one of the collected constraints and solves the resulting constraint system to obtain a new logical input map $I'$ that is similar to $I$ but (likely) leads the execution through a different path. It then sets $I = I'$ and repeats the process.

Conceptually, CUTE executes the code under test both concretely and symbolically at the same time. The actual CUTE implementation first instruments the source code under test, adding functions that perform the symbolic execution. CUTE then repeatedly executes the instrumented code only concretely.

3.1 Logical Input Map

CUTE keeps track of input memory graphs as a logical input map $I$ that maps logical addresses to values that are either logical addresses or primitive values. This map symbolically represents the input memory graph at the beginning of an execution. The reason that CUTE introduces logical addresses is that actual concrete addresses of dynamically allocated cells may change in different executions. Also, the concrete addresses themselves are not necessary to represent memory graphs; it suffices to know how the cells are connected. Finally, CUTE attempts to make consecutive inputs similar, and this can be done with logical addresses. If CUTE used the actual physical addresses, it would depend on malloc and free (to return the same addresses) and more importantly, it would need to handle destructive updates of the input by the code under test: after CUTE generates one input, the code changes it, and CUTE would need to know what changed to reconstruct the next input.

Let $N$ be the set of natural numbers and $V$ be the set of all primitive values. Then, $I : N \rightarrow N \cup V$. The values in the domain and the range of $I$ belong to the set $N$ represents the logical addresses. We also assume that each logical address $l \in N$ has a type associated with it. A type can be $T \ast$ (a pointer of type $T$) (where $T$ can be primitive type or struct type) or $T_p$ (a primitive type). The function $\text{typeOf}(l)$ returns this type. Let the function $\text{sizeOf}(T)$ returns the number of memory cells that an object of type $T$ uses. If $\text{typeOf}(l)$ is $T \ast$ and $I(l) \neq \text{NULL}$, then the sequence $I(v), \ldots, I(v + n - 1)$ stores the value of the object pointed by the logical address $l$ (each element in the sequence represents the content of each cell of the object in order), where $v = I(l)$ and $n = \text{sizeOf}(T)$. This representation of a logical input map essentially gives a simple way to serialize a memory graph.

We illustrate logical inputs on an example. Recall the example Input 3 from Figure 1. CUTE represents this input with the following logical input: $(3, 1, 3, 0)$, where logical addresses range from 1 to 4. The first value 3 corresponds to the value of $p$: it points to the location with logical address 3. The second value 1 corresponds to $x$. The third value corresponds to $p->v$ and the fourth to $p->next$ ($0$ represents NULL). This logical input encodes a set of concrete inputs that have the same underlying graph but reside at different concrete addresses. Similarly, the logical input map for Input 4 from Figure 1 is $(3, 1, 3, 3)$.

3.2 Units and Program Model

A unit under test can have several functions. CUTE requires the user to select one of them as the entry function for which CUTE generates inputs. This function in turn can call other functions in the unit as well as functions that are not in the unit (e.g., library functions). The entry function takes as input a memory graph, a set of all memory locations reachable from the input pointers. We assume that the unit operates only on this input, i.e., the unit has no external functions (that would, for example, simulate an interactive input from the user or file reading). However, a program can allocate additional memory, and the execution then operates on some locations that were not reachable in the initial state. Given an entry function, CUTE generates a main function that first initializes all the arguments of the function by calling the primitive function input() (described next) and then calls the entry function with these arguments. The unit along with the main function forms a closed program that CUTE instruments and tests.

We describe how CUTE works for a simple C-like language shown in Figure 2. START represents the first statement of a program under test. Each statement has an optional label. The program can get input using the expression input(). For simplicity of description, we assume that a program gets all
the inputs at the beginning of an execution and the number of inputs is fixed. CUTE uses the CIL framework [20] to convert more complex statements (with no function calls) into this simplified form by introducing temporary variables. For example, CIL converts \( \text{scanf} \) into \( \text{scanf}(\text{input}) \), where \( \text{scanf} \) is a variable, \( v \) is a constant.

Details of handling of function calls using a symbolic stack are discussed in Section 4.

The C expression \( \&v \) denotes the address of the variable \( v \), and \( *v \) denotes the value of the address stored in \( v \). In concrete state, each address stores a value that either is primitive or represents another memory address (pointer).

3.3 Instrumentation

To test a program \( P \), CUTE tries to explore all execution paths of \( P \). To explore all paths, CUTE first instruments the program under test. Then, it repeatedly runs the instrumented program \( P \) as follows:

```
// input: \( P \) is the instrumented program to test
// \( \text{depth} \) is the depth of bounded DFS
// run_CUTE(\( P \), \( \text{depth} \))
\( T = \{ \}; \ h = (\text{number of arguments in} \( P \)) + 1; \)
\( \text{completed} = \text{false}; \ \text{branch_hist} = \{ \}; \)
\( \text{while not completed} \)
\( \text{execute} \ P \)
```

Before starting the execution loop, CUTE initializes the logical input map \( \mathcal{I} \) to an empty map and the variable \( h \) representing the next available logical address to the number of arguments to the instrumented program plus one. (CUTE gives a logical address to each argument at the very beginning.) The integer variable \( \text{depth} \) specifies the depth in the bounded DFS described in Section 3.4.

Figure 3 shows the code that CUTE adds during instrumentation. The expressions enclosed in double quotes (“e”) represent syntactic objects. We describe the instrumentation for function calls in Section 4. In the following section, we describe the various global variables and procedures that CUTE inserts.

3.4 Concolic Execution

Recall that a program instrumented by CUTE runs concretely and at the same time performs symbolic computation through the instrumented function calls. The symbolic execution follows the path taken by the concrete execution and replaces with the concrete value any symbolic expression that cannot be handled by our constraint solver.

An instrumented program maintains at the runtime two symbolic states \( \mathcal{A} \) and \( \mathcal{P} \), where \( \mathcal{A} \) maps memory locations to symbolic arithmetic expressions, and \( \mathcal{P} \) maps memory locations to symbolic pointer expressions. The symbolic arithmetic expressions in CUTE are linear, i.e. of the form \( a_1 x_1 + \ldots + a_n x_n + c \), where \( n \geq 1 \), each \( x_i \) is a symbolic variable, each \( a_i \) is an integer constant, and \( c \) is an integer constant. Note that \( n \) must be greater than 0. Otherwise, the expression is a constant, and CUTE does not keep constant expressions in \( \mathcal{A} \), because it keeps \( \mathcal{A} \) small: if a symbolic expression is constant, its value can be obtained from the concrete state. The arithmetic constraints are of the form \( a_1 x_1 + \ldots + a_n x_n + c \geq 0 \), where \( \geq \in \{<,\leq,\geq,=,\neq\} \).

The pointer expressions are simpler: each is of the form \( x_p \), where \( x_p \) is a symbolic variable, or the constant NULL. The pointer constraints are of the form \( x_{p_{<}} \geq 0 \). The pointer constraints are of the form \( x_{p_{<}} \geq 0 \).

Given any map \( M \) (e.g., \( \mathcal{A} \) or \( \mathcal{P} \)), we use \( M' = M[m \mapsto v] \) to denote the map that is the same as \( M \) except that \( M'(m) = v \). We use \( M' = M - m \) to denote the map that is the same as \( M \) except that \( M'(m) = \text{undefined} \). We say \( m \in \text{domain}(M) \) if \( M(m) \) is defined.

Input Initialization using Logical Input Map

Figure 4 shows the procedure \( \text{initInput}(\mathcal{I}, l) \) that uses the logical input map \( \mathcal{I} \) to initialize the memory location \( m \), to update the symbolic states \( \mathcal{A} \) and \( \mathcal{P} \), and to update the input map \( \mathcal{I} \) with new mappings.

\( M \) maps logical addresses to physical addresses of memory cells already allocated in an execution, and \( \text{malloc}(n) \) allocates \( n \) fresh cells for an object of size \( n \) and returns the addresses of these cells as a sequence. The global variable \( h \) keeps track of the next unused logical address available for a newly allocated object.

For a logical address \( l \) passed as an argument to \( \text{initInput} \), \( \mathcal{I}(l) \) can be undefined in two cases: (1) in the first execution when \( \mathcal{I} \) is the empty map, and (2) when \( l \) is some logical address that got allocated in the process of initialization. If \( \mathcal{I}(l) \) is undefined and if typeOf(\( l \)) is not a pointer, then the content of the memory is initialized randomly; otherwise, if the typeOf(\( l \)) is a pointer, then the contents of \( l \) and \( m \) are both initialized to NULL. Note that CUTE does not attempt to generate random pointer graphs but assigns all new pointers to NULL. If typeOf(\( \mathcal{I}(l) \)) is a pointer to \( \mathcal{T} \) (i.e., \( \mathcal{T} \ast \) and \( M(l) \) is defined, then we know that the object pointed by the logical address \( l \) is already allocated and we simply initialize the content of \( m \) by \( M(l) \)). Otherwise, we allocate sufficient physical memory for the object pointed by \( \ast \) using malloc and initialize them recursively. In the
Figure 4: Input initialization

Symbolic Execution

Figure 5 shows the pseudo-code for the symbolic manipulations done by the procedure execute_symbolic which is inserted by CUTE in the program under test during instrumentation. The procedure execute_symbolic(m, e) evaluates the expression e symbolically and maps it to the memory location m in the appropriate symbolic state.

Recall that CUTE replaces a symbolic expression that the CUTE’s constraint solver cannot handle with the concrete value from the execution. Assume, for instance, that the solver can solve only linear constraints. In particular, when a symbolic expression becomes non-linear, as in the multiplication of two non-constant sub-expressions, CUTE simplifies the symbolic expression by replacing one of the sub-expressions by its current concrete value (see line L in Figure 5). Similarly, if the statement is for instance \( v'' = v + v' \) (see line D in Figure 5), and both v and v' are symbolic, CUTE removes the memory location &v'' from both A and P to reflect the fact that the symbolic value for v'' is undefined.

Figure 6 shows the function evaluate_predicate(p, b) that symbolically evaluates p and updates path_c. In case of pointers, CUTE only considers predicates of the form \( x = y \), \( x \neq y \), \( x = \text{NULL} \), and \( x \neq \text{NULL} \), where x and y are symbolic pointer variables. We discuss this in Section 3.7. If a symbolic predicate expression is constant, then true or false is returned.

At the time symbolic evaluation of predicates in the procedure evaluate_predicate, symbolic predicate expressions from branching points are collected in the array path_c. At the end of the execution, path_c[0 . . i - 1], where i is the number of conditional statements of P that CUTE executes, contains all predicates whose conjunction holds for the execution path.

Note that in both the procedures execute_symbolic and evaluate_predicate, we skip symbolic execution if the number of predicates executed so far (recorded in the global variable i) becomes greater than the parameter depth, which gives the depth of bounded DFS described next.

Bounded Depth-First Search

To explore paths in the execution tree, CUTE implements a (bounded) depth-first strategy (bounded DFS). In the bounded DFS, each run (except the first) is executed with the help of a record of the conditional statements (which is the array branch_hist) executed in the previous run. The procedure cmp_n_set_branch_hist in Figure 7 checks whether the current execution path matches the one predicted at the end of the previous execution and represented in the variable branch_hist. We observed in our experiments that the execution almost always follows a prediction of the outcome of a conditional. However, it could happen that a prediction is not fulfilled because CUTE approximates, when necessary, symbolic expressions with concrete values (as explained in Section 3.4), and the constraint solver could then produce a solution that changes the outcome of some earlier branch. (Note that even when there is an approximation, the solution does not necessarily change the outcome.) If it ever happens that a prediction is not fulfilled, an exception is raised to restart run_CUTE with a fresh random input.
Given a path constraint \( C \) (see section 3.6). For example, if we use an invariant to evaluate it, our solver is built on top of \( \text{lp}\_\text{solve} \) [18], a constraint solver for linear arithmetic constraints. Our solver provides three important optimizations for path constraints:

**OPT 1** Fast unsatisfiability check: The solver checks if the last constraint is syntactically the negation of any preceding constraint; if it is, the solver does not need to invoke the expensive semantic check. (Experimental results show that this optimization reduces the number of semantic checks by 60-95%.)

**OPT 2** Common sub-constraints elimination: The solver identifies and eliminates common arithmetic sub-constraints before passing them to the \( \text{lp}\_\text{solve} \). (This simple optimization, along with the next one, is significant in practice as it can reduce the number of sub-constraints by 64% to 90%).

**OPT 3** Incremental solving: The solver identifies dependency between sub-constraints and exploits it to solve the constraints faster and keep the solutions similar. We explain this optimization in detail.

Given a predicate \( p \) in \( C \), we define \( \text{vars}(p) \) to be the set of all symbolic variables that appear in \( p \). Given two predicates \( p \) and \( p' \) in \( C \), we say that \( p \) and \( p' \) are dependent if one of the following conditions holds:

1. \( \text{vars}(p) \cap \text{vars}(p') \neq \emptyset \), or
2. there exists a predicate \( p'' \) in \( C \) such that \( p \) and \( p'' \) are dependent and \( p' \) and \( p'' \) are dependent.

Two predicates are independent if they are not dependent.

The following is an important observation about the path constraints \( C \) and \( C' \) from two consecutive concolic executions: \( C' \) differ in the small number of predicates (more precisely, only in the last predicate when there is no backtracking), and thus their respective solutions \( I \) and \( I' \) must agree on many mappings. Our solver exploits this observation to provide more efficient, incremental constraint solving. The solver collects all the predicates in \( C \) that are dependent on \( \neg \text{path}_c[j] \). Let this set of predicates be \( D \). Note that all predicates in \( D \) are either linear arithmetic predicates or pointer predicates, because no predicate in \( C \) contains both arithmetic symbolic variables and pointer symbolic variables. The solver then finds a solution \( I'' \) for the conjunction of all predicates from \( D \). The input for the next run is then \( I = \{I'\} \) which is the same as \( I \) except that for every \( l \) for which \( I'(l) \) is defined, \( I''(l) = I''(l) \). In practice, we have found that the size of \( D \) is almost one-eighth the size of \( C \) on average. If all predicates in \( D \) are linear arithmetic predicates, then \( \text{CUTE} \) uses integer linear programming to compute \( I'' \). If all predicates in \( D \) are pointer predicates, then \( \text{CUTE} \) uses the following procedure to compute \( I'' \):

1. Consider only pointer constraints, which are either equalities or disequalities. The solver first builds an equivalence graph based on (dis)equalities (similar to checking satisfiability in theory of equality [2]) and then based on this graph, assigns values to pointers. The values assigned to the pointers can be a logical address in the domain of \( \text{path}_c[l] \), the constant \( \text{NULL} \) (a special constant), or the constant \( \text{NULL} \) (represented by 0). The solver views \( \text{NULL} \) as a
Generating Inputs with Call Sequences:

One approach to generating data structures is to use sequences of function calls. Each data structure implements functions for several basic operations such as creating an empty structure, adding an element to the structure, removing an element from the structure, and checking if an element is in the structure. A sequence of these operations can be used to generate an instance of data structure, e.g., we can create an empty list and add several elements to it. This approach has two requirements [27]: (1) all functions must be available (and thus we cannot test each function in isolation), and (2) all functions must be used in generation: for complex data structures, e.g., red-black trees, there are memory graphs that cannot be constructed through additions only but require removals [27, 30].

Solving Data Structure Invariants:

Another approach to generating data structures is to use the functions that check invariants. Good programming practice suggests that data structures provide such functions. For example, SGLIB [25] (see Section 5.2) is a popular C library for generic data structures that provides such functions. We call these functions repOk [5]. (SGLIB calls them check_consistency.) As an illustration, SGLIB implements operations on doubly linked lists and provides a repOk function that checks if a memory graph is a valid doubly linked list; each repOk function returns true or false to indicate the validity of the input graph.

The main idea of using repOk functions for testing is to solve repOk functions, i.e., generate only the input memory graphs for which repOk returns true [5, 27]. This approach allows modular testing of functions that implement data structure operations (i.e., does not require that all operations be available): all we need for a function under test is a corresponding repOk function. Previous techniques for solving repOk functions include a search that uses purely concrete execution [5] and a search that uses symbolic execution for primitive data but concrete values for pointers [27]. CUTE, in contrast, uses symbolic execution for both primitive data and pointers.

The constraints that CUTE builds and solves for pointers allow it to solve repOk functions asymptotically faster than the fastest previous techniques [5, 27]. Consider, for example, the following check from the invariant for doubly linked list: for each node n, n.next.prev == n. Assume that the solver is building a doubly linked list with \( N \) nodes reachable along the next pointers. Assume also that the solver needs to set the values for the prev pointers. Executing the check once, CUTE finds the exact value for each prev pointer and thus takes \( O(N) \) steps to find the values for all \( N \) prev pointers. In contrast, the previous techniques [5, 27] take \( O(N^2) \) steps as they search for the value for each pointer, trying first the value NULL, then a pointer to the head of the list, then a pointer to the second element and so on.

3.7 Approximations for Scalable Symbolic Execution

CUTE uses simple symbolic expressions for pointers and builds only (dis)quality constraints for pointers. We believe that these constraints, which approximate the exact path condition, are a good trade-off. To exactly track the pointer constraints, it would be necessary to use the theory of arrays/memory with updates and selections [19]. How-

---

Theorem: The functions may have additional preconditions, but we omit them for brevity of discussion; for more details, see [5].
ever, it would make the symbolic execution more expensive and could result in constraints whose solution is intractable. Therefore, CUTE does not use the theory of arrays but handles arrays by concretely instantiating them and making each element of the array a scalar symbolic variable.

It is important to note that, although CUTE uses simple pointer constraints, it still keeps a precise relationship between pointers: the logical input map (through types), maintains a relationship between pointers to structs and their fields and between pointers to arrays and their elements. For example, from the logical input map (3,1,3,0) for Input 3 from Figure 1, CUTE knows that p->next is at the (logical) address 4 because p has value 3, and the field next is at the offset 1 in the struct cell. Indeed, the logical input map allows CUTE to use only simple scalar symbolic variables to represent the memory and still obtain fairly precise constraints.

Finally, we show that CUTE does not keep the exact pointer constraints. Consider for example the code snippet

```c
*p=0; *q=1; if (*p == 1) ERROR
```

that would enable the program to take the "then" branch. This is because the program contains no conditional that can generate the constraint. Analogously, for the code snippet

```c
a[i]=0; a[j]=1; if (a[i]==0) ERROR, CUTE cannot generate i==j.
```

### 4. IMPLEMENTATION

We have implemented the main parts of CUTE in C, following the algorithms from the previous section. To instrument programs under test to simultaneously run both concrete and symbolic executions, we use CIL [20], an OCAML application for parsing and transforming C programs. To solve arithmetic inequalities, the constraint solver of CUTE uses lp.solve [18], a library for integer programming.

#### 4.1 Program Instrumentation:

To instrument the code, CUTE first uses CIL to simplify a C program into the three-address code that closely follows the syntax given in Figure 2. The difference is that an expression e can also be a function call of the form fun_name(e1, . . . , en). After the simplification, CUTE inserts instrumentation code throughout the simplified code for concotic execution at runtime.

Figure 10 shows examples of the code that CUTE adds during instrumentation for function calls and function definition. The procedure push(kv) pushes the symbolic expression for the address kv to a symbolic stack used for passing symbolic arguments during function calls. The reverse procedure pop(kv) pops a symbolic expression from the symbolic stack and assigns it to the address kv.

A difference between CUTE and traditional symbolic execution is that CUTE does not require instrumentatin of the whole program. Calls to uninstrumented functions proceed only with the concrete execution, without symbolic execution. This allows CUTE to handle programs that use binary libraries whose source code is not available.

#### 4.2 CUTE Toolkit:

The CUTE toolkit provides two commands, cutec and cute, for code instrumentation and running of the instrumented code. The toolkit also provides four macros that give the user additional control over the instrumentation.

<table>
<thead>
<tr>
<th>Before Instrumentation</th>
<th>After Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>// function call</td>
<td>push(kv1); . . . push(kv_n);</td>
</tr>
<tr>
<td>v = f(v1, . . . , vn);</td>
<td>v = f(v1, . . . , vn); pop(kv);</td>
</tr>
<tr>
<td>// function def</td>
<td>pop(kv1); . . . pop(kv_n);</td>
</tr>
<tr>
<td>T f(T x1, . . . , T xn)</td>
<td>B; // body</td>
</tr>
<tr>
<td>return v; }</td>
<td>push(kv); return v; }</td>
</tr>
</tbody>
</table>

Figure 10: Code that CUTE’s instrumentation adds for function calls and function definitions.

The command cutec expects a set of C files and a toplevel function; cutec instruments the C files and compiles the instrumented files with a C compiler. cutec assumes that the program starts by calling the toplevel function and that the input to the program consists of the memory graph reachable from the arguments passed to the toplevel function. cutec generates a main function that first initializes the input for the toplevel function and the symbolic state, and then calls the instrumented toplevel function with the generated input. At the end of the execution of the toplevel function, main calls the constraint solver to generate input for the next execution and stores the inputs in a file.

The command cute takes the executable generated by cutec and executes it iteratively until an error is found or full branch coverage is attained or a depth-first search completes. If an error is found, cute invokes a debugger for the user to replay the erroneous execution.

The CUTE library provides the following macros that the user can insert into the C code under test:

1) CUTEC_input(x), allows user to specify that the variable x (of any type, including a pointer) is an input, in addition to the arguments of the toplevel function. This comes handy to replace any external user input, e.g., scanf( ’%d’, &v) by CUTEC_input(v) (which also assigns value to &v).

2) CUTEC_input_array(p, size). This macro is similar to CUTEC_input except that it assumes that p is a pointer to and specified that p points to an array of size size.

3) CUTEC_assume(pred), where pred is some C predicate. This macro allows the execution to proceed if the pred holds. This way we can restrict the input, e.g., the predicate can be a rep0k() call for some data structure.

4) CUTEC_assert(pred). This macro specifies an assertion whose violation is considered an error.

### 5. EXPERIMENTAL EVALUATION

We illustrate two case studies that show how CUTE can detect errors. In the second case study, we also present results that show how CUTE achieves branch coverage of the code under test. We performed all experiments on a Linux machine with a dual 1.7 GHz Intel Xeon processor.

#### 5.1 Data Structures of CUTE

We applied CUTE to test its own data structures. CUTE uses a number of non-standard data structures at runtime, such as cu_linear to represent linear expressions, cu_pointer to represent pointer expressions, cu_depend to represent dependency graphs for path constraints etc. Our goal in this case study was to detect memory leaks in addition to standard errors such as segmentation faults, assertion violation etc. To that end, we used CUTE in conjunction
with valgrind [26]. We discovered a few memory leaks and a couple of segmentation faults that did not show up in other uses of CUTE. This case study is interesting in that we applied CUTE to partly unit test itself and discovered bugs. We briefly describe our experience with testing the cu_linear data structure.

We tested the cu_linear module of CUTE in the depth-first search mode of CUTE along with valgrind. In 537 iterations, CUTE found a memory leak. The following is a snippet of the function cu_linear_add relevant for the memory leak:

```c
cu_linear *
cu_linear_add(cu_linear *cl, cu_linear *c2, int add) {
    cu_linear* ret=(cu_linear*)malloc(sizeof(cu_linear));
    // skipped 18 lines of code
    if(ret->count==0) return NULL;
}
```

If the sum of the two linear expressions passed as arguments becomes constant, the function returns NULL without freeing the memory allocated for the local variable ret. CUTE constructed this scenario automatically at the time of testing. Specifically, CUTE constructed the sequence of function calls `l1=cu_linear_create(0); l1=cu_linear_negate(l1); l1=cu_linear_add(l1,11,12,1);` that exposes the memory leak that valgrind detects.

5.2 SGLIB Library

We also applied CUTE to unit test SGLIB [25] version 1.0.1, a popular, open-source C library for generic data structures. The library has been extensively used to implement the commercial tool Xrefactory. SGLIB consists of a single C header file, `sglib.h`, with about 2000 lines of code consisting only of C macros. This file provides generic implementation of most common algorithms for arrays, lists, sorted lists, doubly linked lists, hash tables, and red-black trees. Using the SGLIB macros, a user can declare and define various operations on data structures of parametric types.

The library and its sample examples provide verifier functions (can be used as rep0k) for each data structure except for hash tables. We used these verifier functions to test the library using the technique of rep0k mentioned in Section 3.6. For hash tables, we invoked a sequence of its function. We used CUTE with bounded depth-first search strategy with bound 50. Figure 11 shows the results of our experiments.

We chose SGLIB as a case study primarily to measure the efficiency of CUTE. As SGLIB is widely used, we did not expect to find bugs. Much to our surprise, we found two bugs in SGLIB using CUTE.

The first bug is a segmentation fault that occurs in the doubly-linked-list library when a non-zero length list is concatenated with another zero-length list. CUTE discovered the bug in 140 iterations (about 1 seconds) in the bounded depth-first search mode. This bug is easy to fix by putting a check on the length of the second list in the concatenation function.

The second bug, which is a more serious one, was found by CUTE in the hash table library in 193 iterations (1 second). Specifically, CUTE constructed the following valid sequence of function calls which gets the library into an infinite loop:

```c
typedef struct ilist { int i; struct ilist *next; } ilist;
ilist *htab[10];
main() {
    struct ilist *e,*e1,*e2, *m;
    sglib_hashed_ilist_init(htab);
    e=(ilist *)malloc(sizeof(ilist)); e->next = 0; e->e=0;
    sglib_hashed_ilist_add_if_not_member(htab,e,m);
    sglib_hashed_ilist_add(htab,e);
    e2=(ilist *)malloc(sizeof(ilist)); e2->next = 0; e2->e=0;
    sglib_hashed_ilist_is_member(htab,e2);
}
```

where ilist is a struct representing an element of the hash table. We reported these bugs to the SGLIB developers, who confirmed that these are indeed bugs.

Figure 11 shows the results for testing SGLIB 1.0.1 with the bounded depth-first strategy. For each data structure and array sorting algorithm that SGLIB implements, we tabulate the time that CUTE took to test the data structure, the number of runs that CUTE made, the number of branches it executed, branch coverage obtained, the number of functions executed, the benefit of optimizations, and the number of bugs found.

The branch coverage in most cases is less than 100%. After investigating the reason for this, we found that the code contains a number of assert statements that were never violated and a number of predicates that are redundant and can be removed from the conditionals.

The last two columns in Figure 11 show the benefit of the three optimizations from Section 3.5. The column OPT 1 gives the average percentage of executions in which the fast unsatisfiability check was successful. It is important to note that the saving in the number of satisfiability checks translates into an even higher relative saving in the satisfiability-checking time because `lp_solve` takes much more time (exponential in number of constraints) to determine that a set of constraints is unsatisfiable than to generate a solution when one exists. For example, for red-black trees and depth-first search, OPT 1 was successful in almost 90% of executions, which means that OPT 1 reduces the number of calls to `lp_solve` an order of magnitude. However, OPT 1 reduces the solving time of `lp_solve` more than two orders of magnitude in this case; in other words, it would be infeasible to run CUTE without OPT 1. The column OPT 2 & 3 gives the average percentage of constraints that CUTE eliminated in each execution due to common sub-expression elimination and incremental solving optimizations. Yet again, this reduction in the size of constraint set translates into a much higher relative reduction in the solving time.

6. RELATED WORK

Automating unit testing is an active area of research. In the last five years, over a dozen of techniques and tools have been proposed that automatically increase test coverage or generate test inputs.

The simplest, and yet often very effective, techniques use random generation of (concrete) test inputs [4, 8, 10, 21, 22]. Some recent tools use bounded-exhaustive concrete execution [5, 12, 29] that tries all values from user-provided domains. These tools can achieve high code coverage, especially for testing data structure implementation. However, they require the user to carefully choose the values in the domains to ensure high coverage.

Tools based on symbolic execution use a variety of approaches—including abstraction-based model checking [1, 3], explicit-state model checking [27], symbolic-sequence explo-
Figure 11: Results for testing SGLIB 1.0.1 with bounded depth-first strategy with depth 50

<table>
<thead>
<tr>
<th>Name</th>
<th>Run time in seconds</th>
<th># of Iterations</th>
<th># of Branches Explored</th>
<th>% of Functions Tested</th>
<th>OPT 1 in %</th>
<th>OPT 2 &amp; 3 in %</th>
<th># of Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Quick Sort</td>
<td>2</td>
<td>732</td>
<td>43</td>
<td>97.73</td>
<td>2</td>
<td>67.80</td>
<td>49.13</td>
</tr>
<tr>
<td>Array Heap Sort</td>
<td>4</td>
<td>1764</td>
<td>36</td>
<td>100.00</td>
<td>2</td>
<td>71.10</td>
<td>46.38</td>
</tr>
<tr>
<td>Linked List</td>
<td>2</td>
<td>570</td>
<td>100</td>
<td>96.15</td>
<td>12</td>
<td>86.93</td>
<td>88.09</td>
</tr>
<tr>
<td>Sorted List</td>
<td>2</td>
<td>1020</td>
<td>110</td>
<td>96.49</td>
<td>11</td>
<td>88.86</td>
<td>80.85</td>
</tr>
<tr>
<td>Doubly Linked List</td>
<td>3</td>
<td>1317</td>
<td>224</td>
<td>99.12</td>
<td>17</td>
<td>86.95</td>
<td>79.38</td>
</tr>
<tr>
<td>Hash Table</td>
<td>1</td>
<td>193</td>
<td>46</td>
<td>85.19</td>
<td>8</td>
<td>97.01</td>
<td>52.94</td>
</tr>
<tr>
<td>Red Black Tree</td>
<td>2629</td>
<td>1,000,000</td>
<td>242</td>
<td>71.18</td>
<td>17</td>
<td>89.65</td>
<td>64.93</td>
</tr>
</tbody>
</table>

Cadar and Engler proposed Execution Generated Testing (EGT) [6] that takes a similar approach to testing as CUTE: it explores different execution paths using a combined symbolic and concrete execution. However, EGT did not consider inputs that are memory graphs or code that has preconditions. Also, EGT and CUTE differ in how they approximate symbolic expressions with concrete values. EGT follows a more traditional approach to symbolic execution and proposes an interesting method that lazily solves the path constraints: EGT starts with only symbolic inputs and tries to execute the code fully symbolically, but if it cannot, EGT solves the current constraints to generate a (partial) concrete input with which the execution proceeds.

CUTE is also related to the prior work that uses backtracking to generate a test input that executes one given path (that may be known to contain a bug) [13,16]. In contrast, CUTE attempts to cover all feasible paths, in a style similar to systematic testing. Moreover, this initial work did not address inputs that are memory graphs. Viswanathan and Gupta [28] recently proposed a technique that generates memory graphs. They also use a specialized symbolic execution (not the exact execution with symbolic arrays) and develop a solver for their constraints. However, they consider one given path, do not consider unknown code segments (e.g., library functions), and do not use a combined concrete execution to generate new test inputs.

7. DISCUSSION

We next discuss the advantages of using CUTE over traditional symbolic execution based testing approaches.

**Pointer casting and arithmetic**

CUTE often has an advantage over static analysis in reasoning about linked data. For example, to determine if two pointers point to the same memory location, CUTE simply checks whether their values are equal and does not require an alias analysis that may be inaccurate in the presence of pointer casting and pointer arithmetic. For example, for the following C program:

```c
struct foo { int i; char c;};
void * memset(void *s,char c,size_t n) {
  for(int i=0;i<n;i++) ((char *)s)[i]=c; return s;
}
bar (struct foo *a) {
  if (a && a->c == 1) {
    memset(a,0,sizeof(struct foo));
    if (a->c != 1) { ERROR; }}}
```

a fully sound static analysis should report that `ERROR` might be reachable. However, such a sound static-analysis tool would be impractical as it would give too many false alarms. More practical tools, such as BLAST [14], report that the code is safe because a standard alias analysis is not able to see that `a->c` has been overwritten. In contrast, CUTE easily finds a precise execution leading to the `ERROR`. This kind of code is often found in C where `memset` is widely used for fast memory initialization.
Library functions with side-effects

The concrete execution of CUTE helps to remove false alarms, especially in the presence of library functions that can have side-effects. In the above code, for example, if the function memset is a library function with no source code available, static-analysis tools have no way to find out how the function can affect the global heap. In such situations, they definitely give false alarms. However, CUTE can tackle the situation as it can see the side-effect while executing the function concretely.

Approximating symbolic values by concrete values

CUTE combines the concrete and symbolic executions to make them co-operate with each other, which helps to handle situations where most symbolic executors would give uncertain results. For example, consider testing the function f in the following C code:

```c
f(int x, int y) {
    return x*x+(x%2); } 

f(int x, int y) { z=g(x); if (y == z) { ERROR; } }
```

A symbolic executor would generate the path constraint y=x*x+(x%2) that is not in a decidable theory. Thus, it cannot say that ERROR is reachable with guarantee. On the other hand, suppose that CUTE starts with the initial inputs x=3,y=4. In the first execution, since CUTE cannot handle the symbolic expression x*x+(x%2), it approximates z by the concrete expression 3*3+(3%2)=10 and proceeds to generate the path constraint y!=10. Therefore, by solving the path constraint CUTE will generate the inputs x=3,y=10 for the next run which will reveal the ERROR.

Black-box library functions

The same situation arises in the above code if g is a library function whose source code is not available. A symbolic executor would generate the path constraint y=g(x) involving uninterpreted function and would give a possible warning. However, CUTE in the same way as before generates an input leading to the ERROR.

Lazy initialization

One can imagine combining symbolic execution with randomization in several ways. CUTE commits to concrete values before the execution. Another approach would be to use full symbolic execution and generate concrete values on demand [27]. However, this approach does not handle black-box library functions, executes slower as it needs to always check if data is initialized, and cannot "recover" from bad initialization as this example shows:

```c
f(int x){z=x*x+(x%2); if(z==0){ERROR;} if(x==10){ERROR;}}
```

After executing the first if statement, a lazy initializer will initialize x to a random value in any run since it cannot decide the path constraint x*x+(x%2)=8. Thus, it would not be able to take the then branch of the second if. CUTE, however, would generate x=10 for the second run as it simultaneously executes both concretely and symbolically.

8. CONCLUSION

Our work shows that approximate symbolic execution for testing code with dynamic data structures is feasible and scalable. Moreover, we have shown how to efficiently generate dynamic data structures by incrementally adding and removing a node, or by aliasing two pointers. While we described an implementation for C, we have also developed an implementation for the sequential subset of Java. We are currently investigating how to test programs with concurrency using a similar method. We are also investigating the application of the technique to find algebraic security attacks in cryptographic protocols, and security breaches in unsafe languages.

Acknowledgements

We are indebted to Patrice Godefroid and Nils Klarlund for their comments on a previous version of this paper and for suggestions on clarifying the relationship of the current work with DART. Moreover, the first author benefited greatly from interaction with them during a summer internship. We would like to thank Thomas Ball, Cristian Cadar, Sarfraz Khurshid, Alex Orso, Rupak Majumdar, Sameer Sundresh, and Tao Xie for providing valuable comments. This work is supported in part by the ONR Grant N00014-02-1-0715.

9. REFERENCES


